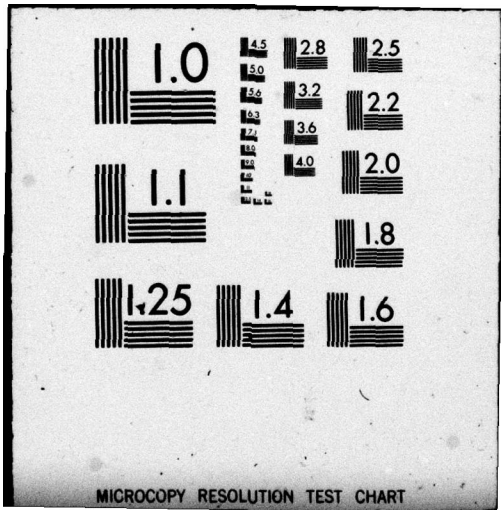
AD-A081 488 GENERAL RESEARCH CORP SANTA BARBARA CA SYSTEMS TECHNO--ETC F/G 9/2
USING EXECUTIVE ASSERTIONS FOR TESTING.(U)
NOV 79 D M ANDREWS, J P BENSON F49620-79-C-0115
UNCLASSIFIED GRC-TM-2282 AFOSR-TR-80-0127 NL

| OF |
AD
A081488

END
DATE
FILMED
4-80
DDC


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6
6.3
7.1
8.0
9.0
MICROCOPY RESOLUTION TEST CHART

Technical Memorandum 2282

Using Executable Assertions for Testing

LEVEL II

by
D. Andrews
J. Benson

November 1979

SYSTEMS TECHNOLOGIES GROUP

GENERAL RESEARCH CORPORATION

A SUBSIDIARY OF FLOW GENERAL INC.

P.O. Box 6770, Santa Barbara, California 93111

Research reported in this document was sponsored by the Air Force Office of Scientific Research (AFSC), United States Air Force, under Contract F49620-79-C-0115.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER: AFOSR-TR-80-0127	2. GOVT ACCESSION NO.: Technical memo.
3. RECIPIENT'S CATALOG NUMBER	
4. TITLE *(and Subtitle)*: USING EXECUTABLE ASSERTIONS FOR TESTING	5. TYPE OF REPORT & PERIOD COVERED: Interim
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s): D. M. Andrews and J. P. Benson	8. CONTRACT OR GRANT NUMBER(s): F49620-79-C-0115
9. PERFORMING ORGANIZATION NAME AND ADDRESS: General Research Corporation, P. O. Box 6770, Santa Barbara, CA 93111	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: 61102F 2304/A2
11. CONTROLLING OFFICE NAME AND ADDRESS: Air Force Office of Scientific Research/NM, Bolling AFB, Washington, D. C. 20332	12. REPORT DATE: November 1979
	13. NUMBER OF PAGES: Five
14. MONITORING AGENCY NAME & ADDRESS *(if different from Controlling Office)*: 8	15. SECURITY CLASS. *(of this report)*: UNCLASSIFIED
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT *(of this Report)*

Approved for public release; distribution unlimited.

GRC-TM-2282

17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20, if different from Report)*

18. SUPPLEMENTARY NOTES

19. KEY WORDS *(Continue on reverse side if necessary and identify by block number)*

20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*

One of the ways of assuring greater reliability of software is to improve testing techniques. Three of the key problems associated with software testing are: choosing adequate test cases, assuring correctness of the results, and reducing the high cost of testing. Some degree of automation is required to help solve these problems. By combining the automated capability of adaptive testing with the use of executable assertions, it is possible to execute a program with a large number of testcases over a wide range of input values. The usual goal of adaptive testing is to maximize some performance value (objective function) for

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE 411 225 UNCLASSIFIED

20. Abstract cont.

the software by automated perturbation of the input parameters. This technique only indirectly leads to locating errors, because of the time-consuming task of examining the usual output from the program. In software testing, the primary goal is to locate the maximum number of errors rather than maximize the performance value. Since software errors can be detected by executable assertions, these assertions can be used to define an objective function for the Adaptive Tester so that a program can be tested automatically and a mapping made of its "error space." A search algorithm is used to generate new test cases based on past performance data about the number of assertion violations. Software testing can become much more efficient and effective through the use of adaptive testing with assertions, because such extensive testing increases the possibility of finding any existing errors and of improving software reliability.

USING EXECUTABLE ASSERTIONS FOR TESTING

D. M. Andrews and J. P. Benson
General Research Corporation
P.O. Box 6770
Santa Barbara, California 93111
805-964-7724 ext. 336

Abstract

One of the ways of assuring greater reliability of software is to improve testing techniques. Three of the key problems associated with software testing are: choosing adequate test cases, assuring correctness of the results, and reducing the high cost of testing. Some degree of automation is required to help solve these problems. By combining the automated capability of adaptive testing with the use of executable assertions, it is possible to execute a program with a large number of testcases over a wide range of input values. The usual goal of adaptive testing is to maximize some performance value (objective function) for the software by automated perturbation of the input parameters. This technique only indirectly leads to locating errors, because of the time-consuming task of examining the usual output from the program. In software testing, the primary goal is to locate the maximum number of errors rather than maximize the performance value. Since software errors can be detected by executable assertions, these assertions can be used to define an objective function for the Adaptive Tester so that a program can be tested automatically and a mapping made of its "error space." A search algorithm is used to generate new test cases based on past performance data about the number of assertion violations. Software testing can become much more efficient and effective through the use of adaptive testing with assertions, because such extensive testing increases the possibility of finding any existing errors and of improving software reliability.

Introduction

Testing is one area of the software development cycle where there is a need for vast improvement. It is frequently the most costly part of the cycle and the most time-consuming. What is needed is not to put more money and time into testing, but to devise a systematic method of exercising a program with a sufficient number of input values over the entire range of possible values in such a way that any existing errors are discovered. To accomplish this goal and to automate as much of the testing process as possible, the techniques developed for adaptive testing are being combined with the use of executable assertions for error detection.

Before software testing can be automated (and become cost effective), two primary problems must be solved: developing adequate test cases to identify errors, and verifying the results of these test cases. The problem of developing test cases has been studied in the Adaptive Verification and Validation research program.[1] The method has been to use various search techniques adapted from optimization theory and artificial intelligence research in order to maximize a performance value (objective function for the software). The original test cases (input values) are either supplied by the tester or developed stochastically. These values are then altered, through a feedback mechanism using heuristics, to maximize the performance value.

In software testing, however, we are not so much interested in maximizing performance as in locating errors. The technique of maximizing a performance function only indirectly helps to locate errors in the software. Errors are usually discovered by examining the output of a program, which is a time-consuming task. Since software errors can be detected through the use of assertions, assertions can be used as the objective function, thus allowing the adaptive testing techniques to be applied to the problem of testing software.

Executable Assertions

Almost any condition or specification can be expressed using executable assertions. An executable assertion is a logical expression which, if evaluated to false, signals the violation of a specification for the program. The logical operators of the assertions have been extended to include the operators of first-order predicate calculus: implication, existence, and universal quantifiers. A preprocessor translates the assertions into executable statements. When the program is executed, the logical expression in each assertion is evaluated. If it is false, an error message is printed which states the name of the module and the line number of the assertion statement. In addition, a tabulation is made of how many times an assertion is violated.

Some examples of assertions (including one which is a call to the logical function OUTCHK) are as follows:

```
INITIAL (VALUE .GE. 0.0 .AND. VALUE .LE. PP*TWOPI)

ASSERT (FM .LE. EA-SIN(EA) + EPS)

ASSERT (.SOME. I IN (1,N)(ARRAY(I) .GE. 0))

FINAL (OUTCHK(MODE,VALUE,ORBEL,STATE))
```

Assertions have many uses and are extremely valuable throughout the entire software cycle. Ideally they should be written during the design phase to state specifications about the variables before any coding takes place. Later, during dynamic tests, these same assertions can be made executable to help in program debugging. Assertions can also be left in the code as a form of documentation, because they can contain much useful information about variables (e.g., the expected range of values or, in distributed systems, path or timing constraints). These built-in specifications are a way of protecting the software during deployment or maintenance from modifications to the code which may alter the expected mode of operations.

Adaptive Testing

Adaptive testing is a technique for identifying how well a program performs in response to changes in its input values. The Adaptive Tester was developed to test the response of simulated ballistic missile defense programs to changes in a threat scenario. The program's performance was defined by the number of reentry vehicles which were not intercepted.

The Adaptive Tester uses the principles of feedback and adaptive search to identify scenarios which result in the maximum tolerable number of reentry vehicles penetrating the defense. An initial scenario is described and input to the test case construction algorithm. This algorithm generates a set of input values for the program being tested. The program is executed and values are recorded which measure its performance. The performance values are then evaluated and compared with past performance values. The change in performance values is then used as input to the adaptive search algorithm which constructs a new scenario. New input data is constructed for this scenario and the program is run again. This cycle continues until scenarios are found which cause the maximum tolerable number of reentry vehicles to penetrate the defense. The input values which characterize each such scenario, and the resulting performance values, define the "performance boundary" of the program.

Methodology For Adaptive Testing With Assertions

The input parameters of a program can be perturbed until areas with maximum errors (indicated by assertion violations) are located. This is done in much the same way as the input parameters are systematically perturbed to degrade the system performance until the performance boundary is reached. The first step, therefore, is to add assertions to the code to be tested, if that has not already been done. The important role of the assertions in this type of testing cannot be emphasized enough: they should be interspersed throughout the code at appropriate places[2], there must be a sufficient number of assertions to monitor each variable, and they must correctly state the performance requirements of the variables. In other words, the success of this type of testing depends on the validity and comprehensiveness of the assertions.

To assure the correctness of the assertions, the second step is to perform preliminary tests of the program with varied input values. A frequent result of this initial testing is that unsuspected errors in the code are uncovered by the assertions. Any errors in the code or the assertions should be corrected before continuing.

The third step is to construct a set of test data by specifying the possible range of values for the input variables. The initial value of a variable is the minimum boundary value and the upper limit (and final value) is the maximum specified value. Next, some interval is chosen through which the value of the variable will be stepped during the testing. In this way, a "grid" of input values is defined over the input space. Figure 1 shows the input space boundaries for a program which was used as part of an experiment to determine if it was feasible to merge the adaptive testing methods with the use of executable assertions for this type of testing. The input values of the transit times were within the range of 0 to 300 microseconds. The probability of any one pulse request in the sequence of requests being a search pulse was allowed to range between 0 and 1.

Once the range of input values is specified and the interval chosen, the rest of the testing is automated. The tester is relieved not only of the chore of choosing new input data but also of the tedious task of wading through reams of output to verify the results of each test case. During the fourth step, the program is executed once for each set of input values and a mapping is made of the error space. The input variables of the program are defined as the independent variables, and the number of assertions which become false when the program is run with a particular set of input values is defined as the dependent variable. Values of the dependent variable define an objective function of errors over the input space. By maximizing this objective function, we can locate the values of the input variables which cause the program to fail.

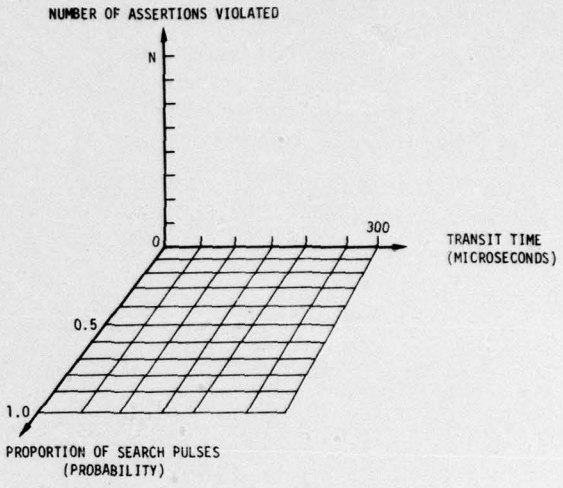


Figure 1. Input Space Boundary

Having defined the error function in much the same way as the performance function is defined for the Adaptive Tester, the fifth step then is to use the search algorithms of the Adaptive Tester to locate values in the program's input variables which cause the most errors to occur. The reason we search for the area of maximum assertion violations is to uncover as many errors as possible. Although some errors will cause several assertions to be violated, other errors are only indicated by a single assertion failure; therefore, it may be necessary for the entire input space be searched for all possible violations. A major assumption in the research is that errors will occur in clusters.

Figure 2 shows the organization of the adaptive testing programs. A test case construction algorithm takes the initial test data and contructs a set of input values with which to run the program.

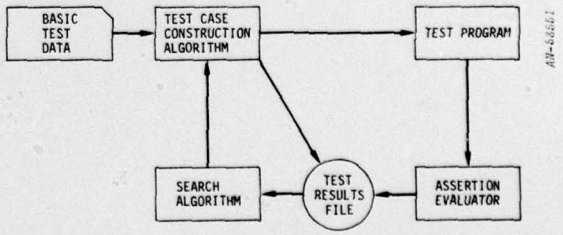


Figure 2. Software for Adaptive Testing Using Assertions

These input data values are also recorded in the test results file for use by the search algorithm in altering the input data. The program is run and the assertion evaluator evaluates the assertions as they are executed and records any assertions that become false. The test results file is then read by the search algorithm, which computes new values for the input variables based upon the assertions violated and the past history of the tests. These new values are input to the test case construction algorithm, which forms a new set of input values for the program and executes it.

Experience With Adaptive Testing Using Assertions

The process that generates a radar schedule in a missile defense simulation was chosen as the test object for a preliminary evaluation of this method of testing software.[3] This set of modules takes a sequence of requests for radar pulses and from these constructs a timing schedule for the radar. The requests may be for search, special search, track, or verify pulses, although for this experiment only search and track pulses were used. Also included in each request is the beam position (direction) in which the pulse should be sent, the desired transmit time, and the length of time between the transmit time and the expected time of the reception of a reflection from the object. This last value is the "transit time" of the pulse. From this sequence of requests, a schedule for the radar is constructed. This schedule must not overlap transmitted pulses with each other or with listening times ("receive windows") and must allow sufficient time between pulses and receive windows to switch beam positions, i.e., "look" in another direction.

In order to show the error space more clearly, two of the input variables were chosen as independent variables in the experiment: the transit time for pulses (i.e., the time between the transmission of the pulse and the receive window) and the number of search pulses in the input sequence. Random input values were generated to simulate a real-time process. The number of assertion violations for each set of input values was tabulated during the testing for use in constructing a three-dimensional grid.

Figure 3 shows the results from testing this program with the set of input values previously illustrated in Figure 1.

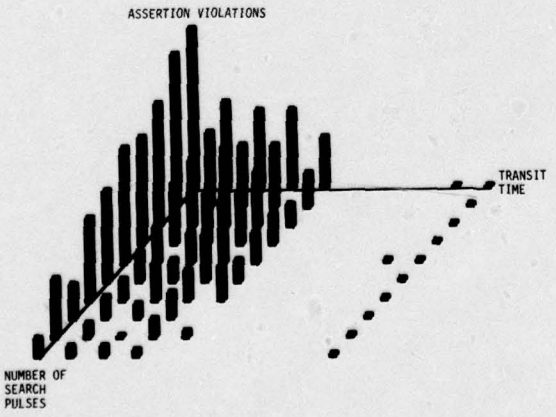


Figure 3. Example of an Error Space Map

These results were generated by varying the mean of the transit times for track pulses from 0 to 300 microseconds in steps of 30 microseconds and by varying the probability of any one pulse's being a search pulse from 0 to 1 in steps of 0.1. The number of assertions violated (for a particular pair of values) is plotted in the vertical axis. The number of assertions violated ranged from a maximum of 20 to a minimum of 0. The assertion violations were the result of latent errors in the program which had not been discovered during all the previous types of testing.

The purpose of this preliminary experiment was to determine what the error space of a program looked like and whether it was feasible to use adaptive search techniques to maximize an error function. The results of this experiment showed that the "error surface" from testing this set of modules was well behaved. It contained maximums, minimums, and gradients which could be used by various search techniques to locate the values in the input space which cause the most errors.

At the conclusion of this first experiment, there were, however, many unanswered question. One, for example, concerns the size of the input space. The range of values for the two input variables was very small, and therefore it was relatively easy to test a large sample of input sequences which vary only slightly. With larger input spaces, sequences of inputs which are very different would have to be generated in order to cover the input space with a reasonable number of test cases.

The main research question to be answered, of course, is whether adaptive search techniques can be applied to the testing of computer programs; that is, whether errors can be located automatically. This question can be broken down into a number of other questions:

1. What does the error function of a typical program look like and is it as well behaved as the one in the preliminary experiment?

   This includes the questions of whether the error space of the program contains singularities, whether it is multi-modal, and whether errors are lumped or randomly distributed.

2. What search methods are applicable to locating error maxima?

   Depending upon the behavior of the error function, one or more of the search methods developed in the adaptive testing research could be applied. In addition, there may be rules of thumb, program-dependent or not, which could be applied to make the search techniques more efficient.

3. Are there other objective functions which can be defined using assertions?

   The objective function formed by merely summing the number of assertions that were violated is a crude one. Other possible ways in which to construct the objective function include weighting the assertions in some way or selecting a subset of the assertions to use.

A second experiment on a much larger scale is now in progress to delve further into the answers to these questions. There are two approaches that could be used to address the question regarding error

spaces: one method would be to examine programs which contained errors, the other would be to seed a program with errors. Since it would be difficult to select a set of programs containing a representative class of errors, the second method seemed preferable because the number and types of errors could be varied. A complex software program which computes orbital elements was selected as the test object. Assertions were added, and a set of errors for "seeding" the program was generated using methods developed by current research.[4] The errors are representative of those found in large programs in both type and frequency of occurrence,[5] and the sites chosen for the seeding were randomly selected.

The input values to the program will be considered two at a time to construct a three-dimensional error space. The values of the other input variables will remain fixed. The error function of the program will be examined by first constructing a coarse grid of input values as in the preliminary experiment. Then the chosen search method and adaptive testing software will be used to map the error space of the program in more detail.

From the error functions derived from each two-variable case, a complete error map will be constructed for the program. The error space map will give a good indication of the shape and characteristics of the error function of the program. The shape of the error function will determine which search algorithms are most applicable to locating the input values of the program which lead to the most errors. The performance of the search algorithm chosen for the experiment will be evaluated.

In regard to the question of applicability of search methods, the most useful algorithm for this application of the Adaptive Tester appears to be the "complex search" which was invented by Box[6] as a method for solving problems with nonlinear objective functions subject to nonlinear inequality constraints. The technique is as follows. Choose a set of points at random and determine the value of the objective function at each one. Then replace the point with the worst performance value with another point which lies on a line formed by the rejected point and the centroid of the remaining points. A set of coefficients is then calculated to determine the exact location of the new point. These coefficients determine the degree of reflection, expansion, shrinkage, contraction, and rotation to be applied in forming the new set of points. The set of random points (N + 1 if there are N dimensions to the problem) is called a complex. New points are selected for the complex using the above technique until a solution is found. The advantage of this technique is that it is somewhat immune to plateaus, discontinuities, and surface irregularities.

As an initial step toward studying the third question, the objective function has been constructed in a different way for the second experiment. This time the tabulation of assertion violations will contain not only the total number of assertion violations but also how many different assertions are violated. The distinction is that the number of total violations may represent one assertion that is in a loop and is violated over and over, or it may represent the violation of many assertions. By keeping track of the number of different assertions violated, it is possible to determine whether more than one assertion (and therefore most likely more than one error) is involved in the total violations. This provides a further refinement of the objective function.

Conclusion

The current experiment will provide a second evaluation of the effectiveness of using executable assertions for software testing. The results will give an indication of the properties of the error function (e.g. continuity) for a typical computer program which contains typical errors. Not only is the error function of the program being explored, but for each sample of errors the efficiency of the search method is being evaluated. The resulting performance of the search algorithm chosen for this experiment will suggest other search methods which could be used to examine the error space. In addition, the performance of the adaptive search technique in locating the maximum value of the error function will indicate how much automation is possible in the testing of computer programs.

Acknowledgement

This research was sponsored by the Air Force Office of Scientific Research (AFSC), United States Air Force, under Contract F49620-79-C-0115. The United States Government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright notation hereon.

References

1. D. W. Cooper, "Adaptive Testing," Second International Conference on Software Engineering, 13-15 October 1976, San Francisco, CA.

2. D. M. Andrews, "Using Executable Assertions for Testing and Fault Tolerance", The Ninth Annual International Symposium on Fault-Tolerant Computing, 20-22 June 1979, Madison, WI.

3. J. Benson, S. Saib, "A Software Quality Assurance Experiment," Software Quality Assurance Workshop, 15-16 November 1978, San Diego, CA.

4. C. Gannon, R. Meeson, N. Brooks, "An Experimental Evaluation of Software Testing," Final Report, General Research Corporation, CR-1-854, May 1979.

5. T. A. Thayer et al., Software Reliability Study, TRW Defense and Space Systems Group RADC-TR76-238, Redondo Beach, CA, August 1976.

6. M. J. Box, "A New Method of Constrained Optimization and Comparison with Other Methods," Computer Journal, Vol. 8 (1965), pp. 42-52.